Unit 2 Review

15-110 – Wednesday 10/28

Agenda

- Unit 2 Overview
- Big-O Runtimes
- Hash Functions
- Recursion
- BFS / DFS

[Recitations will review P vs NP, Tractability, Merge Sort, and more!]

Unit 2 Overview

Unit 2 Goals

Our second unit had two major goals: use various **data structures** to organize data, and calculate the **efficiency** of a variety of algorithms.

How did the topics we discussed fit into these themes?

Data Structures

We discussed several different ways to organize data in a data structure. All of these use **references** when copying data, to let us change the values in the structure directly, and to save space.

- **Lists** let us store data sequentially, and in multiple dimensions if needed
- **Dictionaries** let us store key-value pairs
- **Trees** store hierarchical data using **recursion**, implemented as a nested dictionary
- **Graphs** store connected data, implemented as a dictionary mapping nodes to lists of neighbors

Efficiency

We also discussed the **efficiency** of different algorithms. We abstracted away questions of computer power and specific implementation with the concept of **Big-O notation**, which represents an algorithm's function family.

We discussed a set of **search algorithms** over varying data structures. This included **linear search** for lists, **binary search** for lists and BSTs, **hashed search** for dictionaries, and **BFS/DFS** for graphs.

We also discussed a set of **sorting algorithms** for lists. **Selection sort** and **insertion sort** were intuitive, but not fast; **merge sort** was more complicated, but also more efficient.

Finally, we discussed how the **tractability** of an algorithm influences its ability to run in 'reasonable' time on large sets of data.

Upcoming Topics

In the next unit, we'll talk about how to deal with inefficient algorithms by **scaling up** the amount of computing power used to run algorithms.

We'll then move on to a unit where we address how to apply computer science to **other domains**.

Big-O Runtimes

Big-O Essentials: What to Count?

When measuring the Big-O complexity of an algorithm, we have to specify what it is we’re counting. Some popular choices:

- comparisons: `target == lst[i]`
- assignments: `y[i+1] = x[i]`
- recursive calls: `recSearch(tree["left"], target)`
- all 'actions' in the program (all of the above)

Big-O Essentials: Find the Dominant Term

- We don’t care about linear coefficients.
  - An algorithms that makes 3n comparisons is considered just as fast as an algorithm that makes 2n comparisons: both are O(n).

- Only the dominant term matters:
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Big-O Essentials: Mind the Exponent

O(n^k) is polynomial in n and considered tractable

O(k^n) is exponential in n and considered “slow” (intractable)

Big-O Essentials: Look for a Pattern

Any algorithm that processes each element once is O(n).

- Add up the elements of a list.
- Sum the numbers from 1 to N.
- See if a list contains an odd number, or find the index of the first odd number, or count the odd numbers, or sum all the odd numbers.

Big-O Essentials: When is an algorithm O(n^2)?

Doing an O(n) operation on every element of a list means the total operations is O(n^2).

Common example: nested for loops that are both O(n):

```
for i in range(len(lst)):
    for j in range(i+1, len(lst)):
        if lst[i] == lst[j]:
            print(lst[i], "is duplicated")
```

When is an algorithm O(n^2)?

Suppose we're counting comparisons.

```
for i in range(len(lst)-1):
    if lst[i] in lst[i+1:]:
        print(lst[i], "is duplicated")
```

No nested for loop, but the "in" test is itself O(n), and it's inside a for loop that is O(n), so the algorithm is O(n^2).

When is an algorithm O(log n)?

If we cut the problem size in half each time, and only consider one of the halves, we can make $\log_2(n)$ such cuts, so the algorithm is O(log n).

- Binary search cuts the list in half each time.

Suppose we want the first digit of a long number:

```
while n > 9:
    n = n // 10
```

This code makes $\log_{10}(n)$ divisions, so it's O(log n).

When is an algorithm $O(2^n)$?

If we have a recursive algorithm operating on an input of size n, and each call makes two recursive calls of size n-1, then the algorithm is $O(2^n)$.

```
def allStrings(n,s):
    if n == 0:
        print(s)
    else:
        allStrings(n-1, s+"a")   # first recursive call
        allStrings(n-1, s+"b")   # second recursive call
```

When is an algorithm $O(2^n)$?

If we have a recursive algorithm, and each call produces a result twice as long as the previous result, then the algorithm is $O(2^n)$.

```
def allSubsets(lst):
    if lst == [ ]:
        return [ lst ]
    else:
        result = [ ]
        subsets = allSubsets(lst[1:])
        for s in subsets:
            result.append(s)
            result.append([lst[0]] + s)
        return result
```

Hash Functions

Big Idea

Why do we care about hash functions?

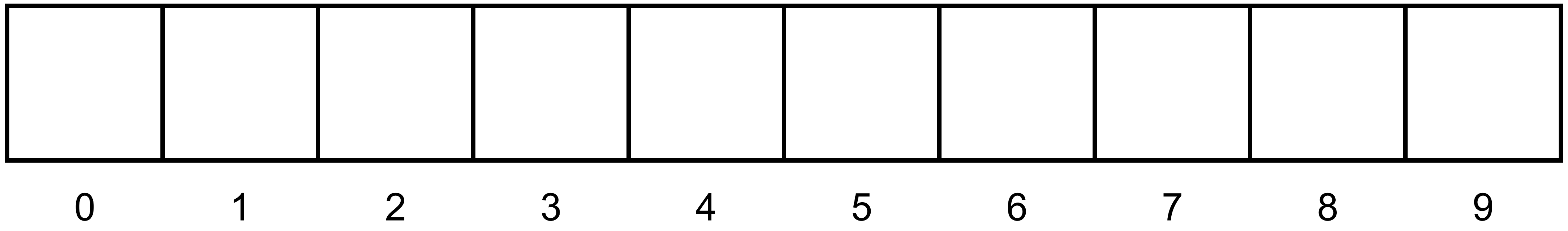
We search all the time, so we want the fastest possible search. Storing items in a hashtable lets us look up whether or not an item is in the table in **O(1)** time. You can't get faster than that!

How can we search in constant time? The algorithm needs to know **where** the value it's looking for will be stored, **if** that value is actually in the table.

Hashtables

A **hashtable** is like a big, empty list of a designated size. Like in a list, each slot ('bucket') in the table is associated with an **integer index**, from 0 to len(table)-1.

When we want to put a value in the hashtable, we insert it at a specific index based on the result of a **hash function**.



Hash Functions

A **hash function** is a function that maps Python values to integers. Those integers can then be used to find an index in the hashtable to store the value.

We can use the built-in Python hash() function, or write our own. Either way, the hash function must follow two rules:

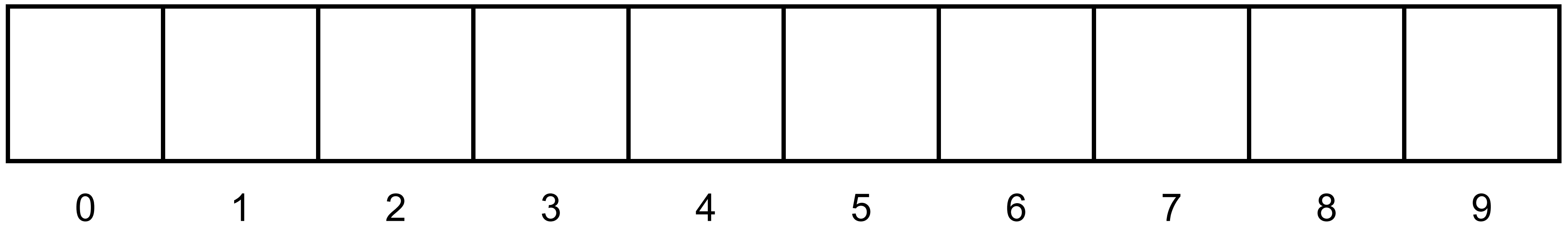
- The value returned when hash is called on x must not change across calls
- The function should usually return different numbers when called on different values

Storing/Findings Values in Hashtables

Both storing a value in a hashtable and checking whether a value is in a hashtable follow the same procedure to get the index to check.

1. Run the hash function on the value to get the hashed value.
2. If the hashed value is larger than the hashtable size, mod it by the hashtable size

Let's practice with some strings and the built-in hash function.



Why O(1)?

Why is looking up a value in a hashtable O(1) time?

We don't need to check every bucket in the hashtable. Only look in one bucket- the one with the index associated with the hashed value.

Important: this only works if the value we're searching for can't change (**immutable**), and if the hashtable is large enough for the stored values to spread out.

Recursion

Big Idea

The core idea of recursion is that we can solve problems by **delegating** most of the work, instead of solving it immediately.

This works because we make the input to the problem **slightly smaller** every time the function is called. That means it will eventually hit a **base case**, where the answer is known right away.

Once the base case returns a value, all the recursive calls can start returning their own values up the **call stack**, until they reach the initial call.

Multiple Recursive Calls

We've done a lot of simple recursive programming in this class, but when we use recursion in the real world, we generally do it by making **multiple recursive calls** in the recursive case. This lets us solve problems that don't fit nicely into loops.

Consider the problem Subset Sum, which we discussed in the Tractability lectures. We're given a list and a target, and we need to see if a subset of the list sums to the target. How can we solve this?

Subset Sum Brainstorming

How can we use **delegation** to solve Subset Sum?

If we give the function a list that has **one fewer element**, and the target, it can decide whether there exists a subset that sums to the target among the rest of the elements.

We can use a **second** recursive call to check whether a subset exists **that uses the element we removed**, which sums to the target!

Base cases: when we find a subset that works, or run out of elements

```python
def subsetSum(lst, target, subset):
    if sum(subset) == target: # we found a subset that works!
        return True
    elif len(lst) == 0: # no more elements, so no subset works... for now
        return False
    else:
        smallerList = lst[1:]
        # try a subset with the element
        attempt1Result = subsetSum(smallerList, target, subset + [lst[0]])
        if attempt1Result == True:
            return True # only return if it works- there's another option!
        # and try a subset without it
        attempt2Result = subsetSum(smallerList, target, subset)
        return attempt2Result

print(subsetSum([4, 1, 3, 7, 6, 23, 12], 22, []))
```

BFS / DFS

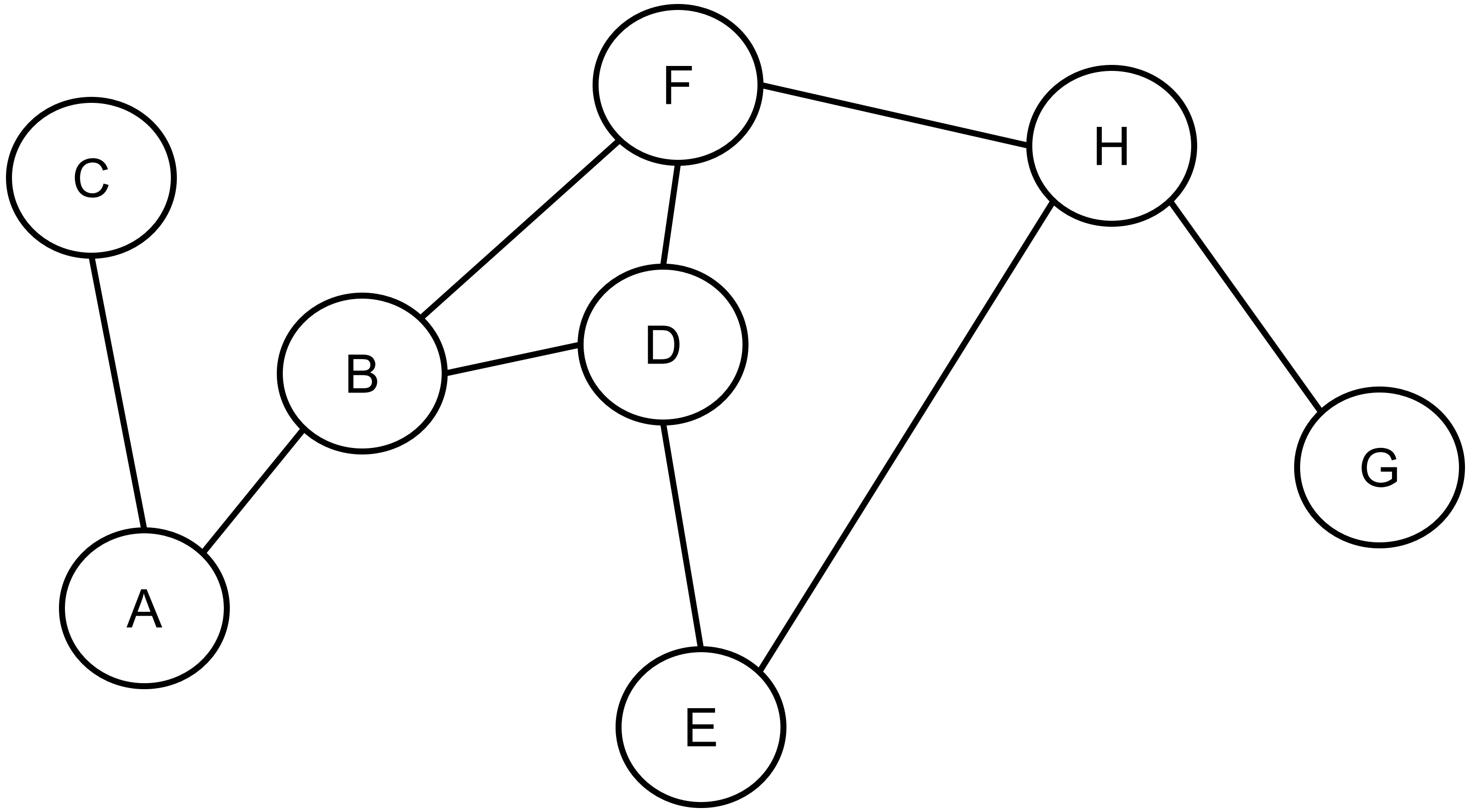
Breadth-First vs Depth-First Search

Both BFS (Breadth-First Search) and DFS (Depth-First Search) share a common goal: they need to find if there's a **path** between a given start node and a given target node in a graph.

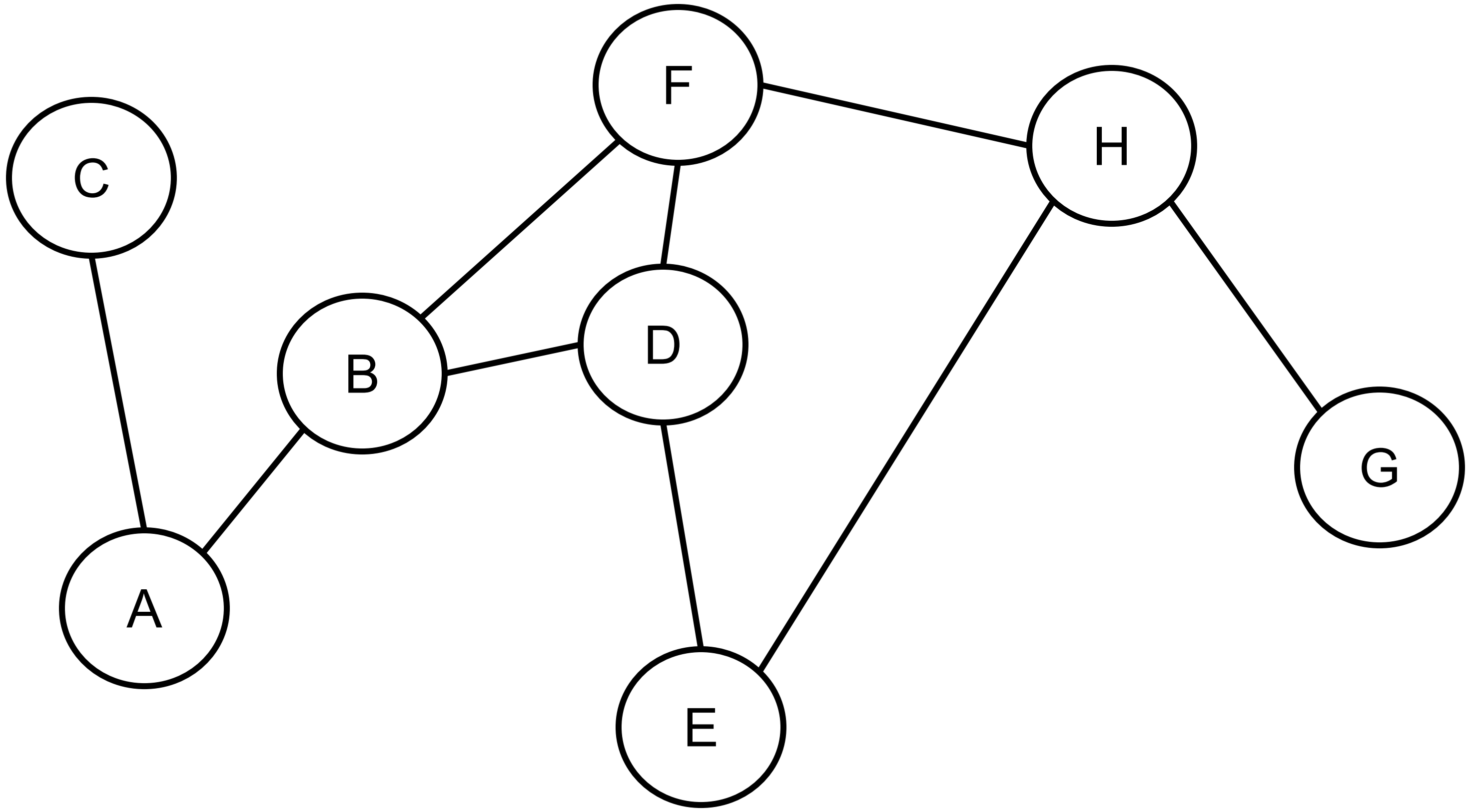
In **BFS**, you start with the nodes connected to the start node and slowly move outwards. It's like how might search for a tiny fallen item- start close to where you're standing, then move outwards.

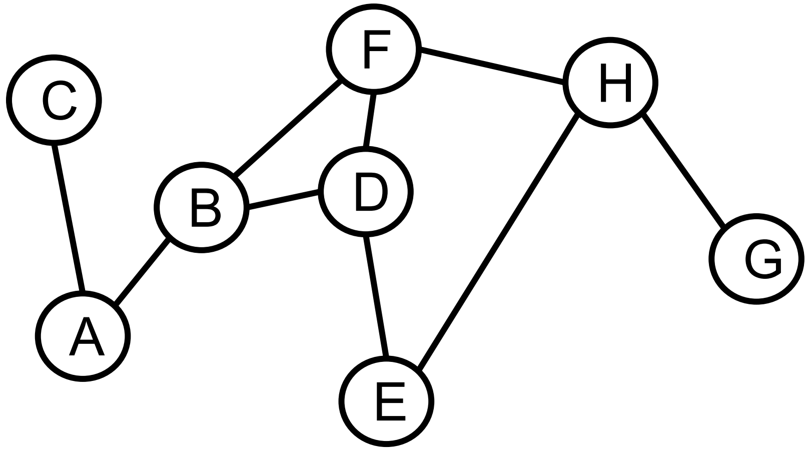
In **DFS**, you go outwards rapidly, backtracking when necessary. It's like how you solve a corn maze- go all the way down one path, then go back when it doesn't work.

BFS Example



DFS Example





Tracing Graph Search Code

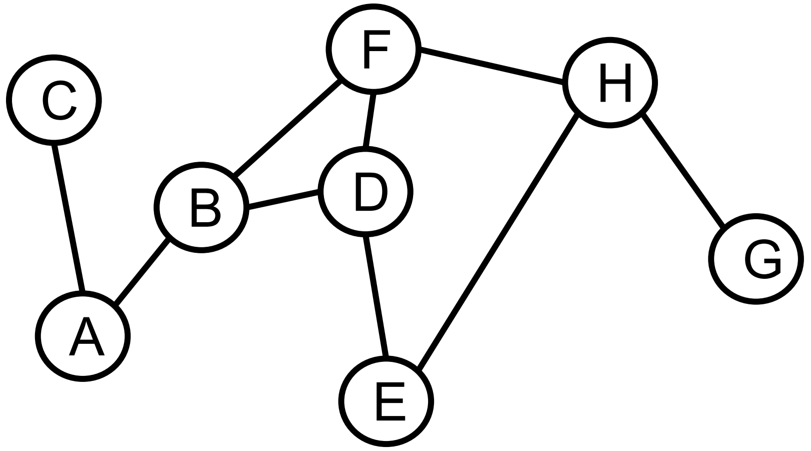
```
def breadthFirstSearch(g, start, target):
    # Set up two lists for visited nodes and to-visit nodes
    visited = [ ]
    frontier = [ start ]

    # Repeat while there are nodes to visit
    while len(frontier) > 0:
        next = frontier[0]
        frontier.pop(0)

        if next == target: # If it's what we're looking for- we're done!
            return True

        # Only expand this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)
            frontier = frontier + g[next]
    return False
```

Tracing Graph Search Code



```
def depthFirstSearch(g, start, target):
    # Set up two lists for visited nodes and to-visit nodes
    visited = [ ]
    frontier = [ start ]

    # Repeat while there are nodes to visit
    while len(frontier) > 0:
        next = frontier[0]
        frontier.pop(0)

        if next == target: # If it's what we're looking for- we're done!
            return True

        # Only check this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)
            frontier = g[next] + frontier
    return False
```